JavaScript 参考 - JavaScript | MDN (mozilla.org)

一、数组封装

1. includes封装：

判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回 false

```
var arr1 = [1, 2, 3];
Array.prototype.includes = function(findItem, start){
    for(var n = 0; n < this.length; n++){
 //循环出来的元素和我们要查找的元素是否相等
    if(this[n]  === findItem){
        return true;//return 之后的代码不会再执行
        }
     }
     return false;
}
   console.log(arr1.includes(2));

//2
Array.prototype.myIncludes = function (val) {
    for (let i = 0; i < this.length; i++) {
        if (this[i] === val) {
            return true;
        }
    }
    return false;
};
数组迭代方
```

2. indexOf封装

返回查找元素的索引

```
//方法1
var arr1 = [1, 2, 3];
Array.prototype.indexOf = function(findItem, start){
    for(var n = 0; n < this.length; n++){
        //循环出来的元素和我们要查找的元素是否相等
```

```
        if(this[n]  === findItem){
            return n;//return 之后的代码不会再执行
        }
    }
    return -1;
 }
 console.log(arr1.indexOf(2));
//方法2
Array.prototype.myIndexOf = function (val, index = 0) {
    if (index < 0) {
        index = -index > this.length ? 0 : index + this.length;
    }
    for (let i = index; i < this.length; i++) {
        if (this[i] === val) {
            return i;
        }
    }
    return -1;
};
```

应用:

```
var arr1 = [1, 2, 3];
Array.prototype.indexOf = function (findItem, start) {
     for (var n = 0; n < this.length; n++) {
         if (this[n] === findItem) {
             return n;
         }
     }
     return -1;
}
    console.log(arr1.indexOf(2));//1
```

3. lastIndexOf封装

```
//1
var arr1 = [1, 2, 3, 2];
Array.prototype.lastIndexOf = function(findItem, start){
```

```
    for(var n = this.length - 1; n > -1; n--){
    //循环出来的元素和我们要查找的元素是否相等
     if(this[n]  === findItem){
          return n;//return 之后的代码不会再执行
         }
     }
      return -1;
 }
        console.log(arr1.lastIndexOf(2));
        console.log(arr1.lastIndexOf(-2));
//2
Array.prototype.myLastIndexOf = function (val, index = this.length) {
    if (index < 0) {
        index = -index >= this.length ? 0 : index + this.length;
    }
    for (let i = index; i >= 0; i--) {
        if (this[i] === val) {
            return i;
        }
    }
    return -1;
};
```

4. isArray封装 （静态方法）

用于确定传递的值是否是一个 Array

```
var arr1 = [1, 2, 3, 2];

Array.isArray = function (arr) {
    return arr.__proto__.constructor === Array;
}

console.log(Array.isArray(arr1));
console.log(Array.isArray('arr1'));
```

5. join封装

```
var arr1 = [1, 2, 3, 2];
Array.prototype.join = function (separ) {
     //如何设置默认值
     if (!separ) {
          separ = ',';
     }
    var str = '';
    for (var i = 0; i < this.length; i++) {
    str += separ + this[i].toString();
    }
    return str.substring(1);
}
console.log(arr1.join('-'));
```

6. filter封装

实现数组元素去重：`filter()`:使用指定的函数测试所有元素，并创建一个包含所有通过测试元素的新数组

```
 //方法1
 var words = ["a", "b", "a", "c"];
 var result = words.filter((word, index) => {
     return words.indexOf(word) === index;
 });
     console.log(result);

//方法2
 let arr = [1, 2, 3, 2, 43, 2, 3, 3, 45, ];
        let newArr = [];
        arr.filter(function (item, ind, arr) {
            if (newArr.indexOf(arr[ind]) === -1) {
                newArr.push(arr[ind]);
            }
        })
        console.log(newArr); //[1, 2, 3, 43, 45]

let arr = [1, 2, 3, 2, 43, 2, 3, 3, 45, ];
```

```
        let newArr = [];
        arr.filter(function (item, ind, arr) {
            if (newArr.indexOf(arr[ind]) === -1) {
                newArr.unshift(arr[ind]);
            }
        })
        console.log(newArr); //[45, 43, 3, 2, 1]

//方法3
let arr1 = [1, 2, 3, 2, 4, 2, 3, 3, 99];
        let arr2 = arr1.filter(function (item, ind, arr) {
            return arr.indexOf(item) === ind;
        })
        console.log(arr2); //[1, 2, 3, 4, 99]

//方法4
function fn(a) {
            let arr3 = a.filter(function (item, ind, arr) {
                return arr.indexOf(item) === ind;
            });
            return arr3;
        }
        let foo = [1, 1, '1', '2', 1];
        let foo1 = fn(foo);
        console.log(foo1); //[1, '1', '2']
```

7. push

- 返回值：新数组的长度
- 思路：数组的长度等于 arguments[i]

```
//push方法应用
let arr = [1, 2, 3, 4, 5, 6];
Array.prototype.myPush = function () {
    for (let i = 0; i < arguments.length; i++) {
        this[this.length] = arguments[i];
        }
    return this.length; //返回新数组的长度
}
```

```
console.log(arr.myPush()); //6
```

8. pop

- 返回值：删除的项（如果空数组，返回 undefined）
- 思路：让数组长度 -1

```
Array.prototype.myPop = function () {
    return this.length == 0 ? undefined : (this[this.length - 1], this.length--);
};
```

9. shift

- 返回值：删除的项
- 思路：让数组前一个值 this[i] 等于后一个值 this[i + 1]，之后把数组长度 -1

```
Array.prototype.myShift = function () {
    if (this.length == 0) {
        return;
    }
    let del = this[0];
    for (let i = 0; i < this.length; i++) {
        this[i] = this[i + 1];
    }
    this.length--;
    return del;
};
```

10. unshift

- 返回值：新数组的长度
- 思路：让数组后一个值 this[i] 等于前 n（n=arguments.length）个值 this[i - arguments.length]，之后把前 n 个值填为 arguments[i]
- ES6 方法实现能简单一些。其实就是拼接数组，之后把拼接的数组一项一项赋值给原数组

```
let arr = [1,2,3,4,5];
Array.prototype.myUnshift = function (...arg) {
    var newArr = [...arg, ...this];
    for (let i = 0; i < newArr.length; i++) {
```

```
        this[i] = newArr[i];
    }
    return this.length;
};
console.log(arr.myUnshift(6)); // 6
```

11. splice

因为 push 实现比较简单，这里用到了 push 方法，能简写一两行

- 返回值：删除的项（数组）
- 思路：按参数数量分别进行判定
  - 参数小于等于1个，从 start 开始添加到新数组，并把添加那项删除（数组长度也减少）
  - 参数大于1个，先进行删除（跟上一步相同）再把数组分成三份，左边+中间被替换的项+右边，合并数组注意：

a. 第一个参数为负数：如果转换为正数，大于数组长度，直接转换为0；小于等于数组长度，需加上数组长度

b. 第二个参数：如果小于0，直接转换为0

c. 第三个参数起：要添加的项

```
Array.prototype.mySplice = function (start, del) {
    let arr = [];
    if (start < 0) {
        start = -start > this.length ? 0 : this.length + start;
    }
    if (arguments.length <= 1) {
        for (let i = start; i < this.length; i++) {
            arr.push(this[i]);
        }
        this.length = start;
    } else {
        del = del < 0 ? 0 : del;
        // 删除数组这一步
        for (let i = 0; i < del; i++) {
            arr.push(this[start + i]);
            this[start + i] = this[start + i + del];
        }
        this.length -= del;
        let lArr = [];
        for (let i = 0; i < start; i++) {
```

```
            lArr.push(this[i]);
        }
        for (let i = 0; i < arguments.length - 2; i++) {
            lArr.push(arguments[i + 2]);
        }
        for (let i = start; i < this.length; i++) {
            lArr.push(this[i]);
        }
        for (let i = 0; i < lArr.length; i++) {
            this[i] = lArr[i];
        }
    }
    return arr;
};
```

- 删除数组那一步，我第一时间想到的是冒泡（把删除项一次一次冒到最后一位），最后出来的代码是下面这样（非常麻烦）

```
let that = del;
for (let i = that; i > 0; i++) {
    if (that <= 0) {
        break;
    }
    for (let j = start; j < this.length - 1; j++) {
        [this[j], this[j + 1]] = [this[j + 1], this[j]];
    }
    arr.push(this[this.length - 1]);
    this.length--;
    that--;
}
```

- 大可不必这样，把删除项的下一位（不删除）往前挪一个一个覆盖要删除的项即可

```
for (let i = 0; i < del; i++) {
    arr.push(this[start + i]);
    this[start + i] = this[start + i + del];
}
this.length -= del;
```

12. concat

- 返回值：拼接后的新数组
- 思路：如果参数是数组，需遍历后一个一个添加到新数组

```
Array.prototype.myConcat = function () {
    let arr = [];
    for (let i = 0; i < this.length; i++) {
        arr[i] = this[i];
    }
    for (let i = 0; i < arguments.length; i++) {
        const el = arguments[i];
        if (Array.isArray(el)) {
            for (let i = 0; i < el.length; i++) {
                arr[arr.length] = el[i];
            }
        } else {
            arr[arr.length] = el;
        }
    }
    return arr;
};
```

13. slice

- 返回值：复制后的新数组
- 思路：如果传参是负数索引，需对其长度进行判定。如果大于数组长度，将其改为0；小于数组长度，将其改为arr.length+(负数索引)

```
Array.prototype.mySlice = function (start = 0, end = this.length) {
    var arr = [];
    if (start < 0) {
        start = -start > this.length ? 0 : this.length + start;
    }
    if (end < 0) {
        end = -end > this.length ? 0 : this.length + end;
    }
    for (let i = start; i < end; i++) {
        arr.push(this[i]);
```

```
    }
    return arr;
};
```

14. flat

- 返回值：扁平后的新数组
- 思路：递归（下面实现没有加上可以指定递归深度的参数，类似arr.flat(Infinity)）

```
Array.prototype.myFlat = function () {
    let arr = [];
    fn(this);
    function fn(ary) {
        for (let i = 0; i < ary.length; i++) {
            const item = ary[i];
            if (Array.isArray(item)) {
                fn(item);
            } else {
                arr.push(item);
            }
        }
    }
    return arr;
};

// 2
Array.prototype.myFlat = function () {
    return this.toString().split(",").map((item) => Number(item));
};
```

15. reverse

- 返回值：倒序后的数组
- 思路：第n个数和倒数第n个数两两对换

```
Array.prototype.myReverse = function () {
    for (let i = 0, j = this.length - 1; j > i; i++, j--) {
        var temp = this[i];
        this[i] = this[j];
```

```
            this[j] = temp;
        }
    }
    return this;
};
```

16. sort

- 返回值：排序后的数组
- 思路：不传参的时候，两两比较 String(xxx) 的值；传参的时候判断 callBack(a-b) 是否大于 0 即可

```
Array.prototype.mySort = function (callBack) {
    if (this.length <= 1) {
        return this;
    }
    if (typeof callBack === "function") {
        for (let i = 0; i < this.length - 1; i++) {
            for (let j = 0; j < this.length - 1 - i; j++) {
                if (callBack(this[j], this[j + 1]) > 0) {
                    [this[j], this[j + 1]] = [this[j + 1], this[j]];
                }
            }
        }
    } else if (typeof callBack === "undefined") {
        for (let i = 0; i < this.length - 1; i++) {
            for (let j = 0; j < this.length - 1 - i; j++) {
                if (String(this[j]) > String(this[j + 1])) {
                    [this[j], this[j + 1]] = [this[j + 1], this[j]];
                }
            }
        }
    } else {
        return "参数异常";
    }
    return this;
};
```

17. forEach

回调函数内部 this 一般指向 window

- 返回值：undefined
- 思路：遍历数组

```
Array.prototype.myForEach = function (callBack) {
    for (let i = 0; i < this.length; i++) {
        callBack(i,this[i]);
    }
};
```

18. map

- 返回值：映射后的新数组
- 思路：遍历数组，把数组每一项经过运算后赋值给新数组

```
Array.prototype.myMap = function (callBack) {
    let arr = [];
    for (let i = 0; i < this.length; i++) {
        let index = i, item = this[i];
        arr[i] = callBack(item, index);
    }
    return arr;
};
```

19. reduce

- 返回值：函数累计处理的结果
- 思路：initial 返回值在数组的每次迭代中被记住，最后成为最终的结果值

```
Array.prototype.myReduce = function (callBack, initial) {
    if (typeof callBack !== "function") throw new TypeError("callBack must be function");
    let i = 0;
    if (typeof initial === "undefined") {
        initial = this[0];
        i = 1;
    }
    for (; i < this.length; i++) {
        initial = callBack(initial, this[i], i)
```

```
    }
    return initial;
};
```

20. find

- 返回值：找到就返回符合的元素，没有返回 undefined
- 思路：遍历数组

```
Array.prototype.myFind = function (callBack) {
    for (let i = 0; i < this.length; i++) {
        if (callBack(this[i], i)) return this[i]
    }
}
```

21. every

- 返回值：只要有一个不符合返回false，如果都符合返回 true
- 思路：遍历数组，一假即假

```
Array.prototype.myEvery = function (callBack) {
    for (let i = 0; i < this.length; i++) {
        if (!callBack(this[i], i)) return false
    }
    return true
}
```

22. some

- 返回值：只要有一个符合就返回 true，如果都不符合返回 false
- 思路：遍历数组，一真即真

```
Array.prototype.mySome = function (callBack) {
    for (let i = 0; i < this.length; i++) {
        if (callBack(this[i], i)) return true
    }
    return false
}
```

23. filter

- 返回值：一个新数组，数组里面是符合条件的所有元素
- 思路：遍历数组

```
Array.prototype.myFilter = function (callBack) {
    if (!Array.isArray(this) || !this.length || typeof callback !== 'function') {
        return []
    }
    let arr = [];
    for (let i = 0; i < this.length; i++) {
        if (!callBack(this[i], i)) {
            arr.push(this[i]);
        }
    }
    return arr
}
```

二、数组去重

1.利用Set()+Array.from()

- Set对象：是值的集合，你可以按照插入的顺序迭代它的元素。 Set中的元素只会出现一次，即Set中的元素是唯一的。
- Array.from() 方法：对一个类似数组或可迭代对象创建一个新的，浅拷贝的数组实例。

```
const result = Array.from(new Set(arr));
    console.log(result) ; // [ 1, 2, 'abc', true, false, undefined, NaN ]
```

> **注意：**以上去方式对NaN和undefined类型去重也是有效的，是因为NaN和undefined都可以被存储在Set中， NaN之间被视为相同的值（尽管在js中：NaN !== NaN）。

2.利用两层循环+数组的splice方法

通过两层循环对数组元素进行逐一比较，然后通过`splice`方法来删除重复的元素。此方法对NaN是无法进行去重的，因为进行比较时NaN !== NaN。

```
function removeDuplicate(arr) {
```

```
  for (let i = 0; i < arr.length; i++) {
      for (let j = i + 1; j < arr.length; j++) {
         if (arr[i] === arr[j]) {
         arr.splice(j, 1)
         len-- // 减少循环次数提高性能
         j-- // 保证j的值自加后不变
       }
     }
   }
    return arr
}
 const result = removeDuplicate(arr)
 console.log(result) // [ 1, 2, 'abc', true, false, undefined, NaN, NaN ]
```

3.利用数组的indexOf方法

indexOf() 方法：返回调用它的`String`对象中第一次出现的指定值的索引，从 `fromIndex` 处进行搜索。如果未找到该值，则返回 -1。

```
function removeDuplicate(arr) {
 const newArr = []
 arr.forEach(item => {
 if(newArr.indexOf(item) === -1) {
     newArr.push(item)
    }
})
 return newArr  ; // 返回一个新数组
}
 const result = removeDuplicate(arr);
console.log(result) ; // [ 1, 2, 'abc', true, false, undefined, NaN, NaN ]
```

4.利用数组的includes方法

此方法逻辑与`indexOf`方法去重异曲同工，只是用`includes`方法来判断是否包含重复元素。

includes()方法：用来判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回 false。

```
```

```
function  removeDuplicate(arr) {
  const newArr = []
      arr.forEach(item => {
        if(!newArr.includes(item)) {
      newArr.push(item)
     }
  })
  return newArr;
}
 const result = removeDuplicate(arr)
 console.log(result) ; // [ 1, 2, 'abc', true, false, undefined, NaN ]
```

> **注意：**为什么includes能够检测到数组中包含NaN，其涉及到includes底层的实现。如下图为includes实现的部分代码，在进行判断是否包含某元素时会调用sameValueZero方法进行比较，如果为NaN，则会使用isNaN()进行转化。

简单测试includes()对NaN的判断：

```
const testArr = [1, 'a', NaN]
console.log(testArr.includes(NaN)) // true
```

5.利用数组的filter()+indexOf()

`filter`方法会对满足条件的元素存放到一个新数组中，结合`indexOf`方法进行判断。

filter() 方法：会创建一个新数组，其包含通过所提供函数实现的测试的所有元素。

```
function removeDuplicate(arr) {

    return arr.filter((item, index) => {

       return arr.indexOf(item) === index
  })
}
 const result = removeDuplicate(arr);
 console.log(result) ;// [ 1, 2, 'abc', true, false, undefined ]
```

> **注意：**这里的输出结果中不包含NaN，是因为indexOf()无法对NaN进行判断，即arr.indexOf(item) === index返回结果为false。测试如下：

```
const testArr = [1, 'a', NaN]
console.log(testArr.indexOf(NaN)) // -1
```

6.利用Map()

Map对象是`JavaScript`提供的一种数据结构，结构为键值对形式，将数组元素作为map的键存入，前端培训然后结合has()和set()方法判断键是否重复。

Map 对象：用于保存键值对，并且能够记住键的原始插入顺序。任何值（对象或者原始值）都可以作为一个键或一个值。

```
function removeDuplicate(arr) {
    const map = new Map()
    const newArr = []
    arr.forEach(item => {
        if(!map.has(item)) {
        // has()用于判断map是否包为item的属性值
        map.set(item, true)
        // 使用set()将item设置到map中，并设置其属性值为true
        newArr.push(item)
        }
    })
    return newArr;
}
 const result = removeDuplicate(arr);
 console.log(result) ;// [ 1, 2, 'abc', true, false, undefined, NaN ]
```

> **注意：**使用Map()也可对NaN去重，原因是Map进行判断时认为NaN是与NaN相等的，剩下所有其它的值是根据 === 运算符的结果判断是否相等。

7.利用对象

其实现思想和`Map()`是差不多的，主要是利用了对象的属性名不可重复这一特性。

```
function removeDuplicate(arr) {
 const newArr = []
 const obj = {}
 arr.forEach(item => {
```

```
  if(!obj[item]) {
     newArr.push(item)
       obj[item] = true
      }
   })
 return newArr;
}

 const result = removeDuplicate(arr);
 console.log(result) ;// [ 1, 2, 'abc', true, false, undefined, NaN ]
```

三、数组封装应用：

```
//方法返回数组中满足提供的测试函数的第一个元素的值
Array.prototype.find = function (fn) {
      console.log('我自己写的');
      for (var i = 0; i < this.length; i++) {
                if (fn(this[i], i, this)) {
                    return this[i];
                }
            }
        }

        Array.prototype.findIndex = function (fn) {
            console.log('我自己写的');
            for (var i = 0; i < this.length; i++) {
                if (fn(this[i], i, this)) {
                    return i;
                }
            }
            return -1;
        }

        var array1 = [5, 12, 8, 130, 44];
        var found = array1.find(function (item) {
            return item % 4 === 0
        });

```

```
26          console.log(found);
27
28          var foundIndex = array1.findIndex(function (item) {
29              return item % 10 === 0
30          });
31
32          console.log(foundIndex);
```